

Guidance on LMS Software Procedures

from the LaRC Software Engineering Process Group (SEPG)

1. Purpose

- 1.1 This document provides guidance on selecting both the life cycle and the development approach for software being developed at the NASA Langley Research Center (LaRC) and on both verifying and validating that software. It supplements the Langley Management System (LMS) software procedures (particularly LMS-CP-5528) by providing additional information that software developers may find useful. However, the LMS procedures are the official procedures that must be used at LaRC, and this document does not supersede anything found in them.

2. Generic Phase Model

- 2.1 The standard way of presenting a life cycle is as a sequence of phases, where each one completes before the next one starts. Each phase has a defined set of outputs against which achievement can be measured. Thus, software life cycle phases provide a natural set of milestones along a project's development. During and at the end of each phase, progress can be monitored, risks can be identified, and estimates can be generated. All of these items can be used to build and refine the plans for the next phase as illustrated by Figure 1.

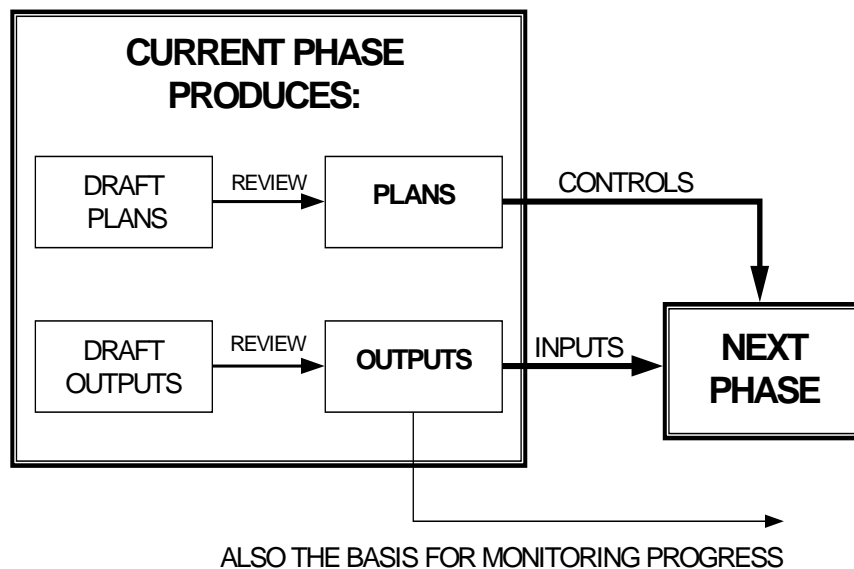


Figure 1. Generic Phase Model.

3. Selecting the Life Cycle Phase Option

3.1 Figure 2 describes the full life cycle phases (Options A and B) as listed in IEEE/EIA 12207.0 [1] and basic tailoring options (C and D). The software life cycle begins when either a software project's "System Requirements" (i.e., Option A) or "User Requirements" (Option B) become available. For example, one might receive a request (User Requirement) to upgrade existing software, or one might receive a request (System Requirement) for the development of new software as part of a larger parent project work package. In either case, the Software Manager should consider the requirements given to the software development team as the Requester Requirements in LMS-CP-5528.

3.2 In many instances, software life cycle phases can be combined. For example, use of a COTS graphics package or a COTS database tool may simplify the software design phase. In this case, the Software Requirements, Architectural Design, and Detailed Design phases could be combined into one design phase (i.e., see Figure 2, Option C). In another example, a trade-off analysis might conclude that the use of a "virtual code" based COTS tool such as LabView can satisfy the majority of the requester requirements. In this case, multiple software phases (Software Requirements, Architectural Design, Detailed Design, Code and Test, and Integration) may be combined as shown in Option D (the minimum life cycle).

3.3 The Software Manager may use the basic life cycle options in Figure 2 and the following explanatory text as guides in tailoring the software life cycle phases to the specific project. Although it is not fully illustrated in Figure 2, all four life cycle phased options can begin with either the Systems Requirements Analysis phase or the User Requirements phase, depending on whether or not there is an encompassing systems-level project.

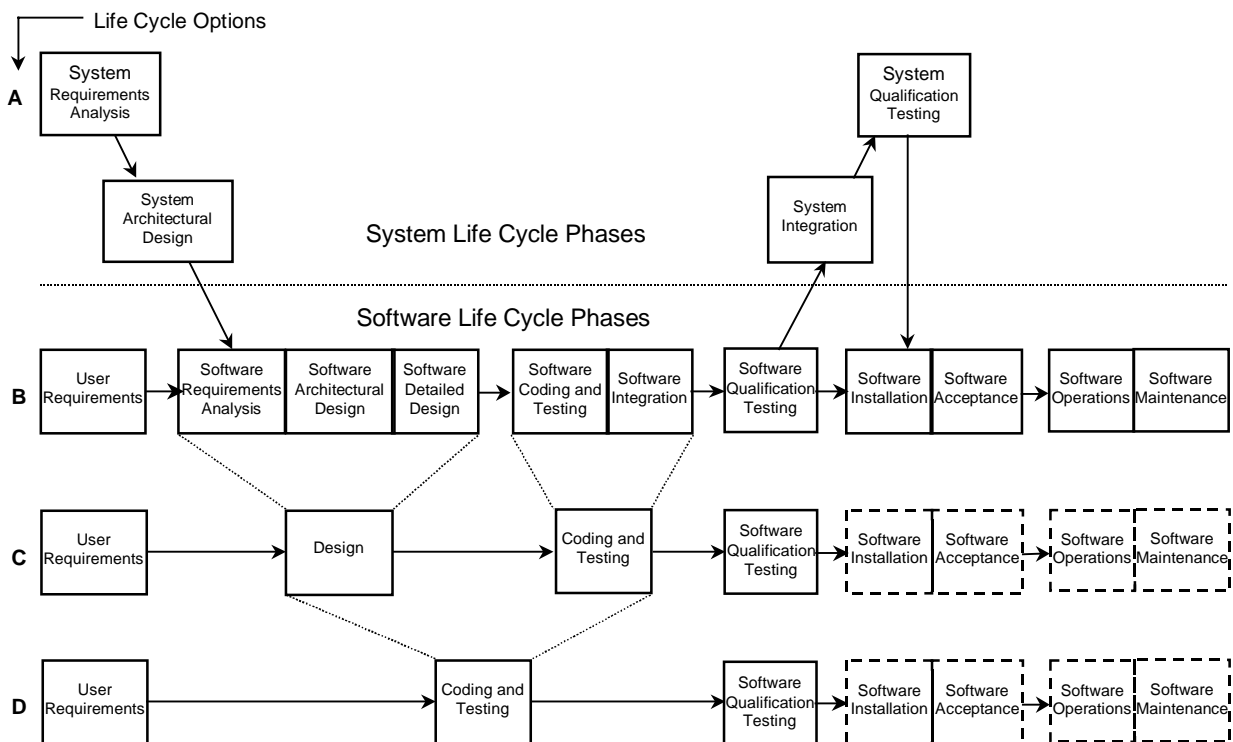


Figure 2. Software life-cycle phase options.

Descriptions of software life-cycle phase options in Figure 2:

Option A: The software project is initiated by the submission of System Requirements. These System Requirements are called Requester Requirements in LMS-CP-5528. In this case, the software supplier supports many of the system level activities.

Option B: The software project is not part of an encompassing system project and starts with the submission of User Requirements. However, the project is large or complex and requires the rigor of the full life cycle. The User Requirements are called Requester Requirements in LMS-CP-5528.

Option C: The software project is initiated by the submission of Requester Requirements (which may be System or User Requirements). These requirements are complete, not complex, and can be mapped directly into a single level of design to define the structure of the code. Because there is no need to distinguish between the logical or abstract software requirements model and the physical or concrete design model generated as part of the architectural design in this example, some phases (Software Requirements Analysis, Software Architectural Design, and Software Detailed Design) are combined into a single design phase. For example, the use of object-oriented methods can minimize the distinction between those phases.

Option D: The software project is initiated by the submission of Requester Requirements. Either the software is very simple and is comprised of no more than a few components, so that translation of requester requirements into code is straightforward, or modifications to existing code are simple and do not require changes in the corresponding software design.

Note: The IEEE document, *The Draft Standard for Developing Software Life Cycle Processes* [2] and the DERA document, *Guide to Tailoring the Project Lifecycle* [3] also provide guidance on defining a life cycle for a project.

- 3.4 In Figure 2, Software Installation, Acceptance, and Operations phases are shown as dashed boxes because they may be omitted if not appropriate (e.g., if the software is not being transferred outside the developing organization or to a different environment for operations, or if an operations phase is not applicable). In some cases, software is created with no intention of reuse or maintenance. For this reason, the maintenance phases are also shown as dashed boxes.
- 3.5 A number of factors influence the choice of life cycle, including size, class, and the development approach. The full life cycle is usually adopted for large software projects and is strongly recommended for medium software projects. For smaller software projects, phases can be combined to suit the simplicity of the work being undertaken. Although the life cycle choice is based on many considerations, a fuller life cycle option may be required by the requester or the parent project. The phases in Figure 2 are illustrated using a 'waterfall' approach. Other approaches are discussed in section 4.
- 3.6 The software life cycle phases shown in Figure 2 are briefly described below. (Note: the descriptions below are summaries of text in IEEE/EIA 12207.0 *Standard for Information Technology—Software Life Cycle Processes* [1].)

System Requirements Analysis: The specific intended use of the system to be developed is analyzed to specify system requirements. The System Requirements specification describes: functions and capabilities of the system; business, organizational and user requirements; safety, security, human-factors engineering, interface, operations, and maintenance requirements; and design constraints and qualification requirements.

System Architectural Design: The top-level architecture of the system is established. The architecture identifies items of hardware, software, and manual-operations. It ensures that all system requirements are allocated among the items.

Software Requirements Analysis: The Software Requirements are documented. These Software Requirements are the interpretation of the Requester Requirements (either System or User) that the software team uses to begin development. They include: functional and capability specifications (including performance, physical characteristics, and environmental conditions); interfaces; qualification requirements; safety and security specifications; data definitions and database requirements; user operation and execution requirements; and maintenance requirements. In some cases, the Requester Requirements and the Software Requirements may be one and the same.

Software Architectural Design: The Software Requirements are transformed into an architecture (and integration plan) that describes the top-level structure and identifies the software components. Each requirement for the software is allocated to one or more software components and is further refined to facilitate detailed design.

Software Detailed Design: A detailed design for each software component is developed. Software components are decomposed into lower level software units that can be coded, compiled, and tested. The Software Requirements are allocated from the software components to the software units. Detailed designs of the interfaces external to the software, between the software components, and between the software units are also documented.

Software Coding and Testing: The software units and databases are developed, coded, compiled, tested, and documented.

Software Integration: The software units and software components are integrated in accordance with the integration plan of the Software Architectural Design; the aggregate is tested to ensure that it satisfies the Software Requirements.

Software Qualification Testing: The implementation of each Requester Requirement is tested for compliance and to ensure that the integrated software is ready for delivery to the user or system.

System Integration: The software configuration items are integrated with the hardware items and manual operations. As they are integrated, the aggregates are tested against their requirements.

System Qualification Testing: Implementation of each System Requirement is tested for compliance and to ensure that the system is ready for delivery.

Software Installation: The software is installed in accordance with the installation plan to ensure that software code and databases initialize, execute, and terminate as specified.

Software Acceptance Support: Acceptance testing and review of the software are performed.

Software Operations: For each release, the software product undergoes operational testing to satisfy the specified criteria prior to the release of the software product for operational use. The system is operated in its intended environment according to the user documentation. Assistance and consultation are provided to the users as requested. User requests for corrections and modification are forwarded to the maintenance processes.

Software Maintenance: Problem reports, modification requests, or requests to migrate software to a new operational environment are analyzed, options for implementing the modifications are considered, and approvals for the selected modifications are sought. The affected documentation and software units are identified and the appropriate life cycle phases listed above are used to implement the modification. When appropriate, software retirement is also performed.

- 3.7 It is recommended that the software manager document the following items in the SPMP: the project control metrics to be collected for each life cycle phase, the collection process, and the responsibilities for collection and analysis. (These metrics are in addition to those collected on the Software Metrics Collection Form and are used specifically for project tracking and control.)

4. Selecting the Software Development Approach

- 4.1 The basic software development life cycles phases are shown in Figure 2 as a sequential (i.e., waterfall) approach, but usually projects are developed using variants of this approach. The following paragraphs provide guidance on choosing an appropriate software development approach, based on project size, complexity, and risk. For further guidance on life cycle approaches, see the *Software Management Guidebook*, Section 5.1 or the *DERA Guide to Tailoring the Project Lifecycle*, page 302.
- 4.2 **Waterfall Development Approach:** In the waterfall software development approach, Figure 3, each phase is completed before the next is started. Each phase produces products that are used in the next phase. This software development approach is appropriate only for very low risk, small projects whose requirements are well documented and understood. It is not appropriate for larger projects or for projects where all the requirements are not known prior to design or where the requirements are likely to change.

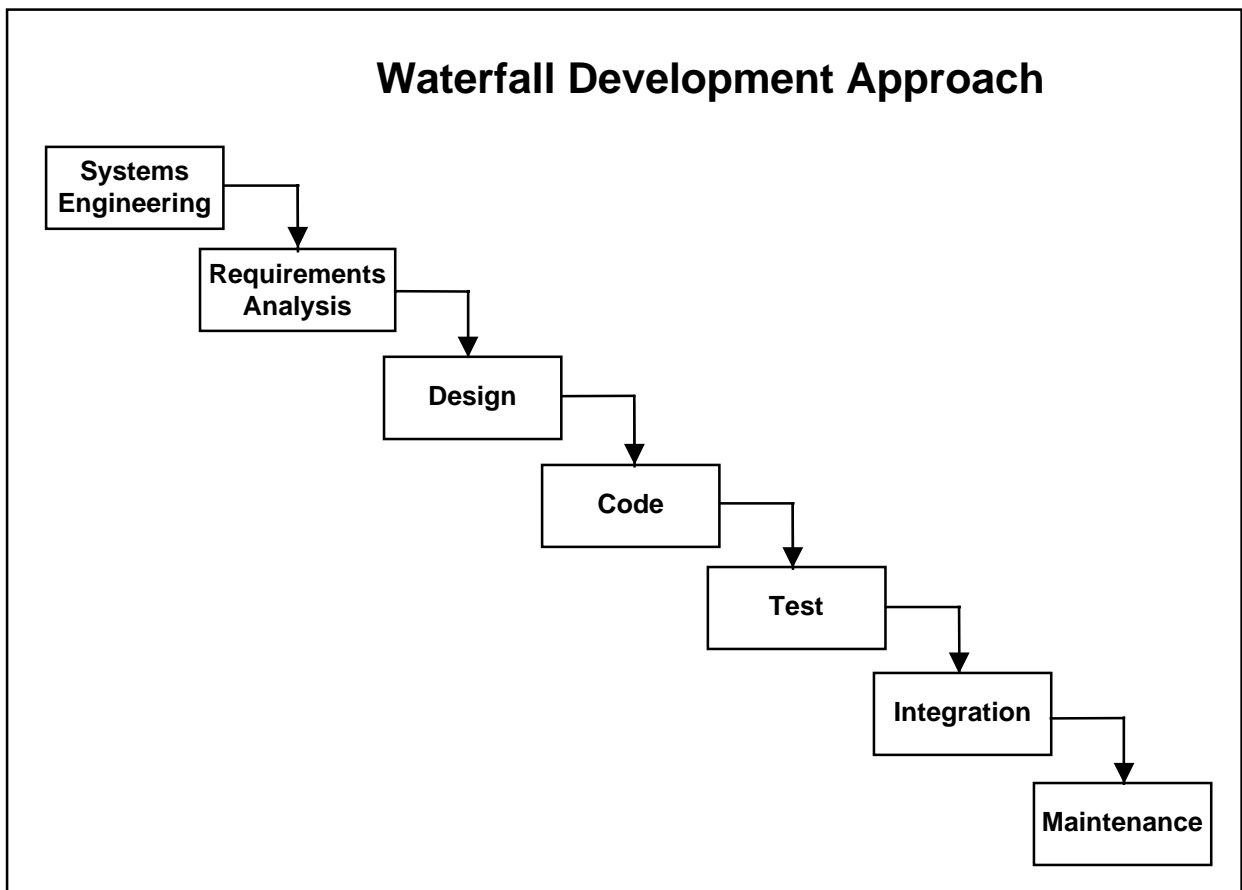


Figure 3. The waterfall development approach.

4.3 Incremental Development Approach: The incremental software development approach is an adaptation of the waterfall approach. It is used for larger projects and projects in which the requirements may change. Under the incremental development approach the whole project is broken into a series of smaller deliverables called builds. Software phases are repeated for each build. The earlier builds can be used to implement the more stable requirements. They can also be used to deal with the more significant risks early in the project and thus increase the probability of project success by eliminating the risks before they become problems. In addition, the higher priority requirements can be implemented in the earlier builds. Therefore, if the project runs over budget and must be terminated before all the requirements are implemented, only the lowest priority functionality is sacrificed. It also provides the opportunity to deliver limited functionality early in the project. This approach also helps simplify the integration phase because only subsets of the software are being integrated at any one time and when errors occur it is easier to pinpoint their location. The errors are usually in the new software set just integrated into the system or in the interface with the new set.

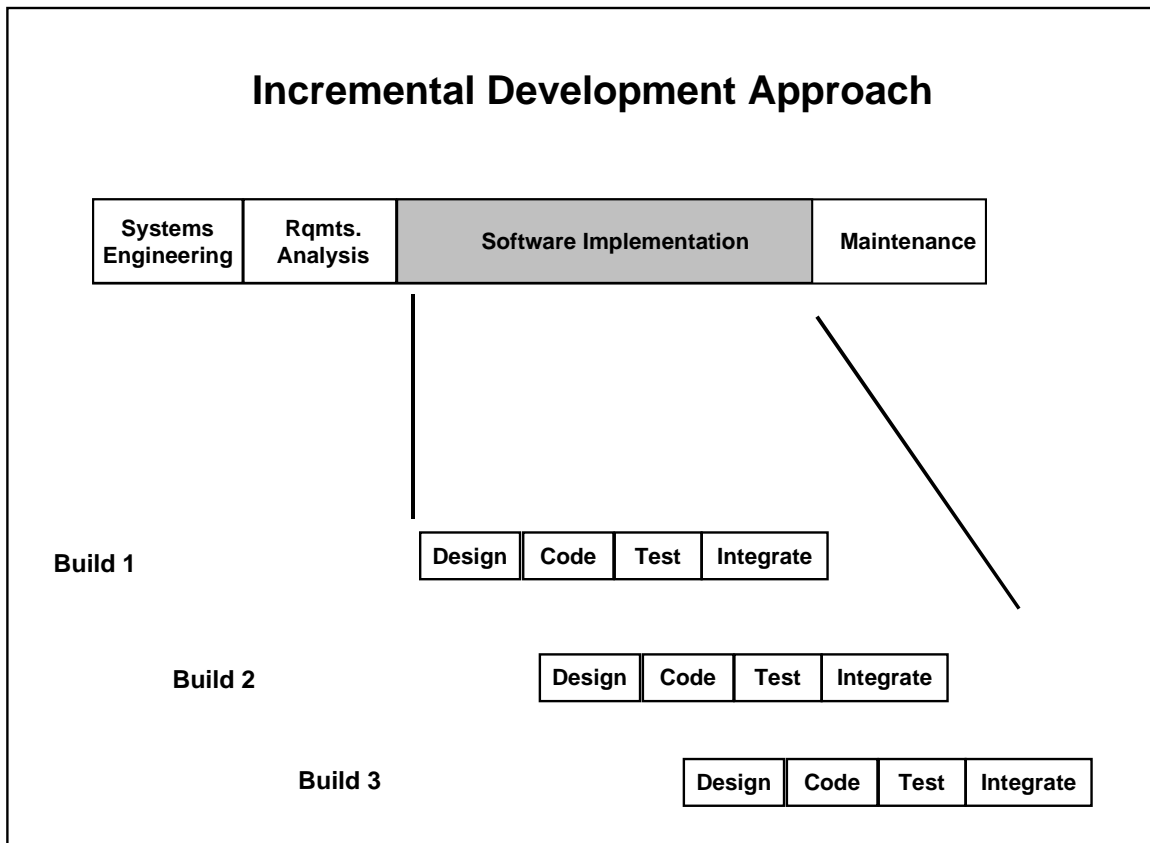


Figure 4. The incremental development approach.

4.4 **Evolutionary Development Approach:** The evolutionary development approach is much like the incremental in which multiple builds are performed. However, in the later builds, requirements can be added based on requester feedback on the earlier builds. The requirements evolve and are updated as better understanding of the requester needs is achieved and issues are resolved. This development approach is used for more complex systems where all the requirements are not known up front.

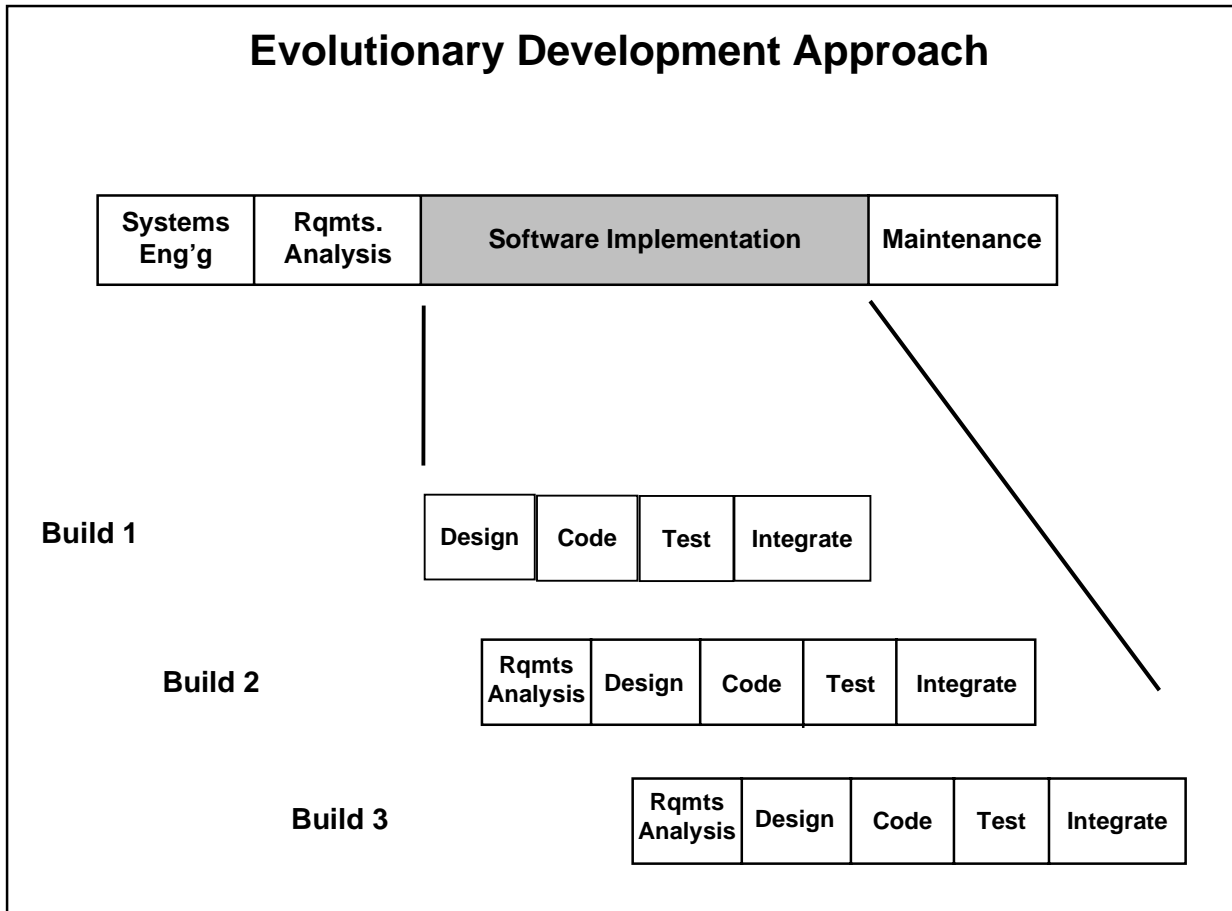


Figure 5. The evolutionary development approach.

4.5 **Spiral Development Approach:** risk management largely drives the spiral software development approach. It recognizes there may be many uncertainties in the project and works to reduce the probability of negative impacts due to those uncertainties. The spiral development approach is a further refinement of the basic waterfall approach. It starts with a series of learning cycles that begin with defining the objectives and then performing a risk identification and analysis to uncover areas of greatest uncertainty in the project. The number of iterations through the cycle varies based on the uncertainty of the problem. Simple problems take one pass and look very much like the standard waterfall approach. More complicated problems may involve several passes through the cycle to resolve uncertainty to an acceptable level. Risk management is performed to identify and analyze risks. Risk mitigation plans identify corrective actions to be taken to reduce the probability and impact of the risks. The mitigation plans are monitored to determine if they removed or reduced the risks as planned. The spiral life cycle approach is appropriate for large, complex systems with very significant risks and systems where all the requirements are not known at the beginning of the project.

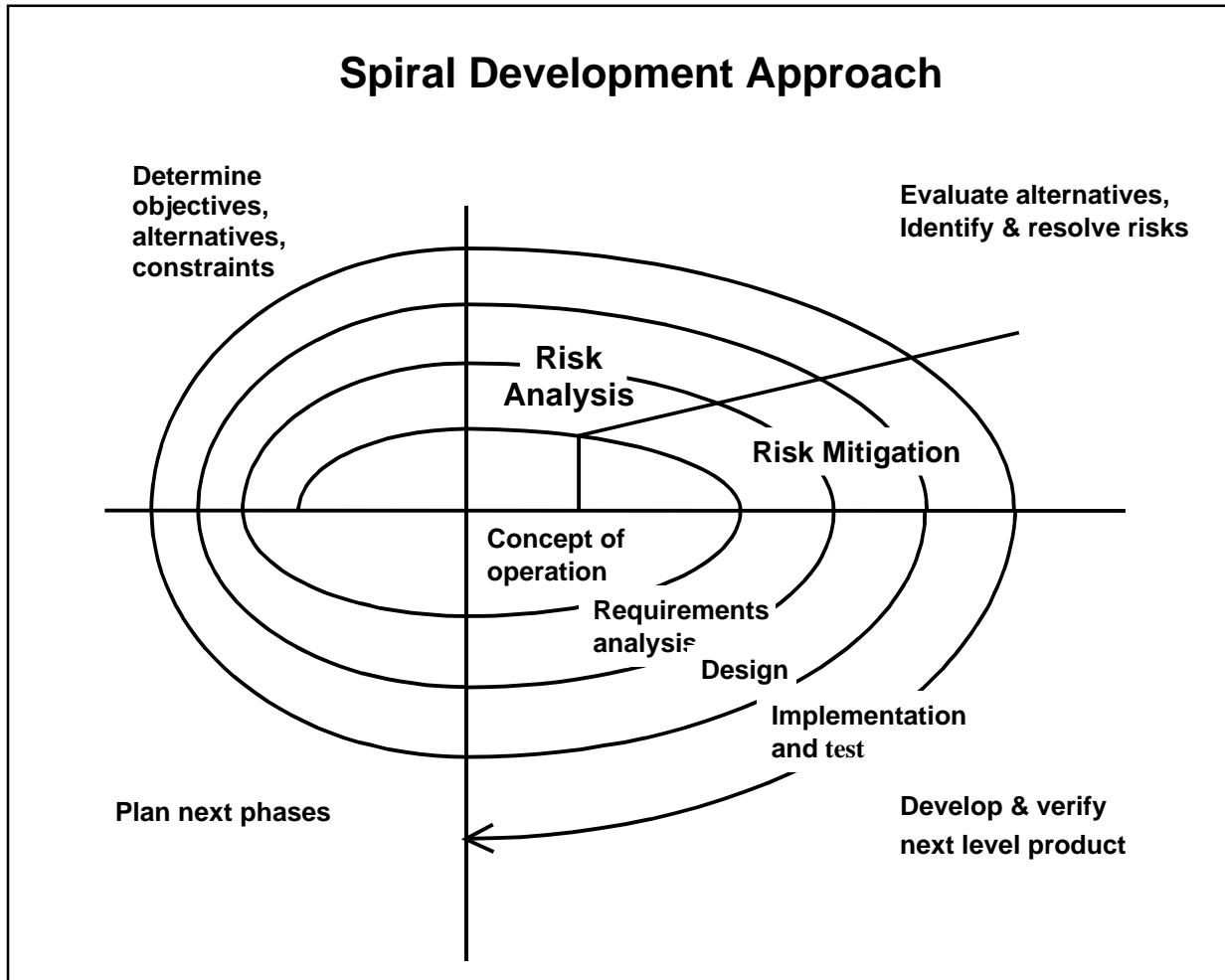


Figure 6. The spiral development approach.

5. Guidance on Verification and Validation

5.1 *Verification* is done to determine whether the products of a given development phase satisfy the conditions imposed at the start of the phase. Verification activities should be conducted throughout the software life cycle to determine whether the products of each phase satisfy the conditions imposed at the beginning of that phase. For example, Software Requirements may be verified for traceability to System or User Requirements; the same requirements may also be verified for testability. Similarly, the software design may be verified against (i.e., checked that it can be derived from) the Software Requirements. Such verifications are critical to the successful completion of any software project. Figure 7 shows the relationships between life cycle phases and the associated verifications. Figure 8 shows the relationships between life cycle phases and the associated verification activities for the minimum life cycle.

5.2 *Validation* activities should be conducted to determine whether a complete system or component satisfies specified system, requester, and software requirements. These requirements are often specified as specific “user” criteria (e.g., “the software must start automatically at 8:00 AM each day, print the pre-specified summary report, and terminate without user intervention”) or as specific “performance” criteria (e.g., “the software must acquire all data at a rate of 100 times per second”). Normally, execution of test cases/procedures is the main activity that validates the software products against their specified requirements. Validation activities should also occur during the Software Acceptance Phase. Figure 7 shows the relationships between life cycle phases and the associated validation activities. Figure 8 shows the relationships between life cycle phases and the associated validation activities for the minimum life cycle.

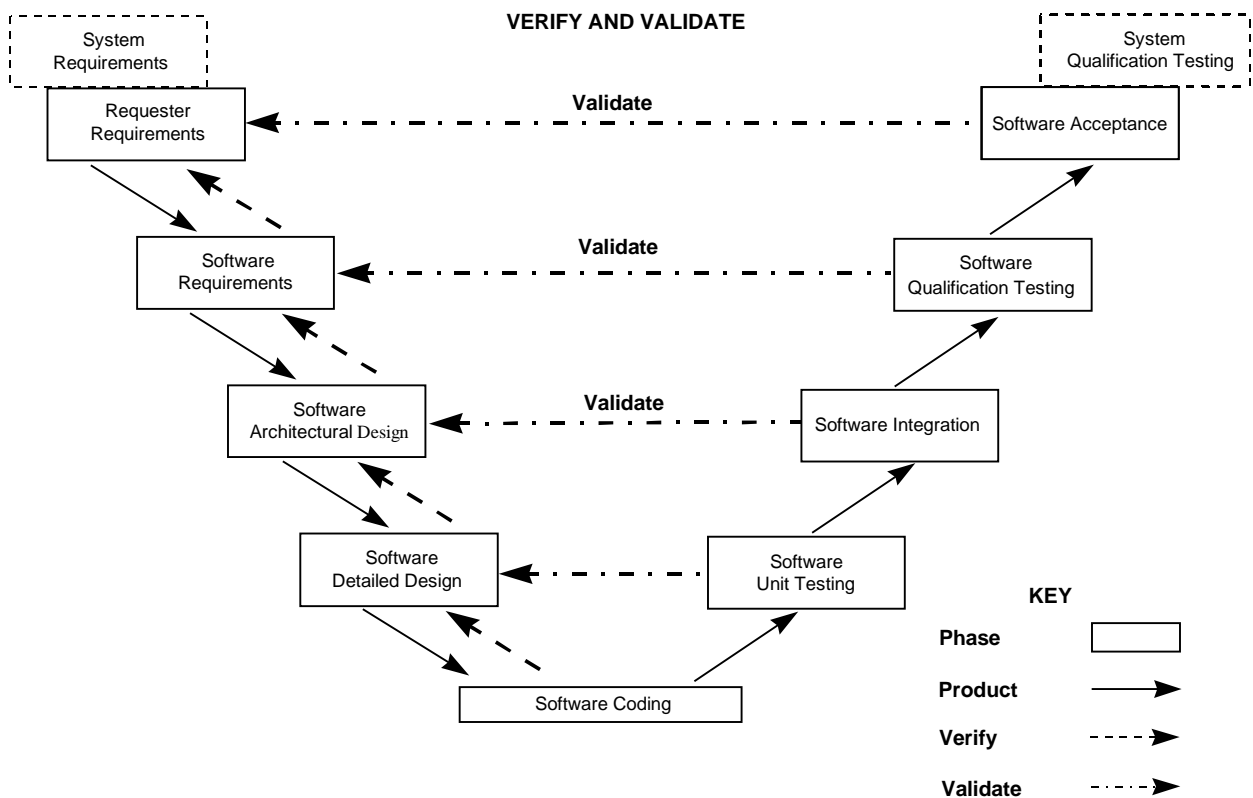


Figure 7. Relationships between life cycle phases, verifications, and validation.

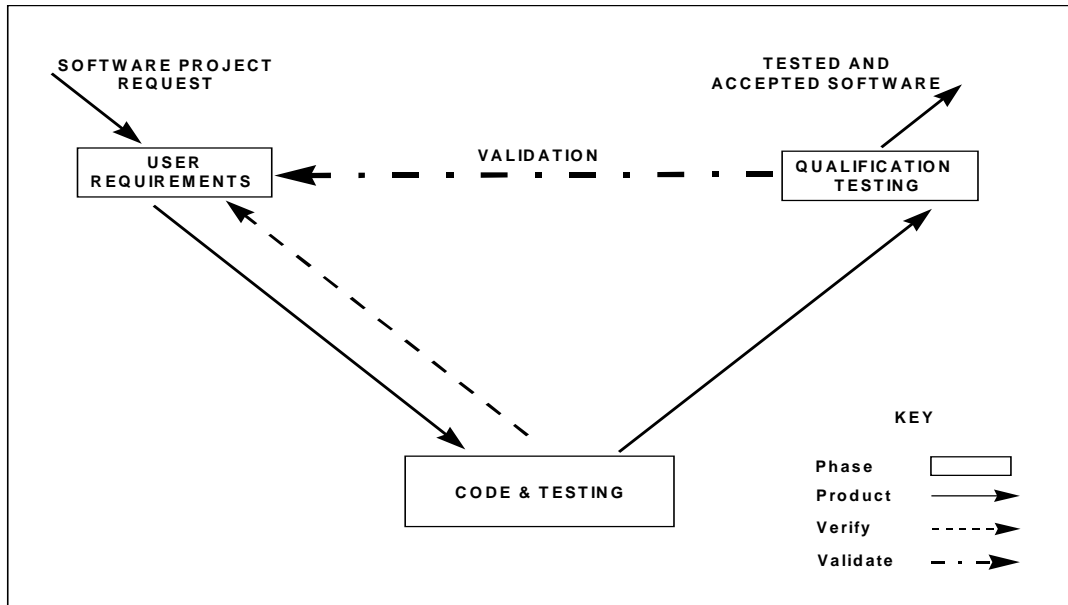


Figure 8. Verification and validation for low-control software using the minimum software life cycle.

5.3 Characteristics of the project (e.g., cost, size, type, complexity, risk, consequences of failure, lifespan, etc.) should be taken into consideration when determining the degree of verification and validation activities for a given project.

6. References

1. IEEE/EIA Standard 12207.0-1996, IEEE/EIA Standard, Industry Implementation of International Standard ISO/IEC 12207: 1995, Standard for Information Technology—Software Life Cycle Processes.
2. IEEE Draft Standard, P1074, D8, Draft Standard for Developing Software Life Cycle Processes, August 1997. (URL: <http://sw-eng.larc.nasa.gov/process/rindex.html>)
3. DERA/COR/G049/1.0, Defence Evaluation and Research Agency, UK, Guide to Tailoring the Project Lifecycle, 1997. (URL: http://www.lerc.nasa.gov/WWW/iso9000/DERA/ebms/cor/bms/corporat/Go49_1.doc)